УСОВЕРШЕНСТВОВАНИЕ БЛОГА ЗА СЧЕТ ПРОДВИНУТЫХ ФУНКЦИОНАЛЬНОСТЕЙ

УСОВЕРШЕНСТВОВАНИЕ БЛОГА ЗА СЧЕТ ПРОДВИНУТЫХ ФУНКЦИОНАЛЬНОСТЕЙ

В данном проекте мы расширим функциональности приложения blog за счет функций, которые в настоящее время можно найти на многих блоговых платформах. В данном проекте будут рассмотрены следующие темы:

- использование канонических URL-адресов для моделей;
- создание дружественных для поисковой оптимизации URL-адресов постов;
- добавление постраничной разбивки в представление списка постов;
- разработка представлений на основе классов;
- отправка электронных писем с помощью Django;
- использование форм Django, позволяющих делиться постами по электронной почте;
- добавление комментариев к постам с использованием форм из моделей.

Все используемые в этом проекте пакеты Python включены в файл requirements.txt

УСОВЕРШЕНСТВОВАНИЕ БЛОГА ЗА СЧЕТ ПРОДВИНУТЫХ ФУНКЦИОНАЛЬНОСТЕЙ

Канонический URL-адрес – это предпочтительный URLадрес ресурса. Его можно представить как URL-адрес наиболее репрезентативной страницы с конкретным контентом. На сайте могут быть разные страницы, которые показывают посты, но есть один URL-адрес, который используется в качестве главного URL-адреса поста.

Канонические URL-адреса позволяют указывать URL-адрес мастер-копии страницы. Django дает возможность в своих собственных моделях реализовывать метод get_absolute_ url(), который возвращает канонический URL-адрес объекта.

УСОВЕРШЕНСТВОВАНИЕ БЛОГА ЗА СЧЕТ ПРОДВИНУТЫХ ФУНКЦИОНАЛЬНОСТЕЙ

Мы будем использовать URL-адрес post_detail, определенный в шаблонах URL-адресов приложения, чтобы формировать канонический URL-адрес для объектов Post.

Django предоставляет различные функции-резольверы URLflhtcjd1 , которые позволяют формировать URL-адреса динамически, используя их имя и любые требуемые параметры. Мы будем использовать функциюутилиту reverse()2 модуля django.urls. Отредактируйте файл models.py приложения blog, импортировав функцию reverse() и добавив метод get_absolute_url() в модель Post, как показано ниже. Новый исходный код выделен жирным шрифтом

1 Резольвер URL-адресов – это программная утилита или функция, которая конвертирует логический адрес или метаданные в физический URL-адрес целевых данных.

2 URL-адрес, или URL-указатель (от англ. Uniform Resource Locator, аббр. URL, т. е. Унифицированный указатель ресурса), указывает на ресурс в сети. Функция reverse в Django выполняет обратное действие и используется для отыскания URL-адреса

УСОВЕРШЕНСТВОВАНИЕ БЛОГА ЗА СЧЕТ ПРОДВИНУТЫХ ФУНКЦИОНАЛЬНОСТЕЙ

Мы будем использовать URL-адрес post_detail, определенный в шаблонах URL-адресов приложения, чтобы формировать канонический URL-адрес для объектов Post.

Django предоставляет различные функции-резольверы URLflhtcjd1 , которые позволяют формировать URL-адреса динамически, используя их имя и любые требуемые параметры. Мы будем использовать функциюутилиту reverse()2 модуля django.urls. Отредактируйте файл models.py приложения blog, импортировав функцию reverse() и добавив метод get_absolute_url() в модель Post, как показано ниже.

1 Резольвер URL-адресов – это программная утилита или функция, которая конвертирует логический адрес или метаданные в физический URL-адрес целевых данных.

2 URL-адрес, или URL-указатель (от англ. Uniform Resource Locator, аббр. URL, т. е. Унифицированный указатель ресурса), указывает на ресурс в сети. Функция reverse в Django выполняет обратное действие и используется для отыскания URL-адреса

УСОВЕРШЕНСТВОВАНИЕ БЛОГА ЗА СЧЕТ ПРОДВИНУТЫХ ФУНКЦИОНАЛЬНОСТЕЙ

Новый исходный код выделен жирным шрифтом

```
from django.db import models
from django.utils import timezone
from django.contrib.auth.models import User
from django.urls import reverse

class PublishedManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset()\
                      .filter(status=Post.Status.PUBLISHED)

class Post(models.Model):

    class Status(models.TextChoices):
        DRAFT = 'DF', 'Draft'
```

УСОВЕРШЕНСТВОВАНИЕ БЛОГА ЗА СЧЕТ ПРОДВИНУТЫХ ФУНКЦИОНАЛЬНОСТЕЙ

```
        PUBLISHED = 'PB', 'Published'

    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250)
    author = models.ForeignKey(User,
                               on_delete=models.CASCADE,
                               related_name='blog_posts')
    body = models.TextField()
    publish = models.DateTimeField(default=timezone.now)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    status = models.CharField(max_length=2,
                              choices=Status.choices,
                              default=Status.DRAFT)

    class Meta:
        ordering = ['-publish']
        indexes = [
            models.Index(fields=['-publish']),
        ]

    def __str__(self):
        return self.title

    def get_absolute_url(self):
        return reverse('blog:post_detail',
                       args=[self.id])
```

Новый исходный код выделен жирным шрифтом

УСОВЕРШЕНСТВОВАНИЕ БЛОГА ЗА СЧЕТ ПРОДВИНУТЫХ ФУНКЦИОНАЛЬНОСТЕЙ

Функция reverse() будет формировать URL-адрес динамически, применяя имя URL-адреса, определенное в шаблонах URL-адресов. Мы использовали именное пространство blog, за которым следуют двоеточие и URL-адрес post_detail.

Напомним, что именное пространство blog определяется в главном файле urls.py проекта при вставке шаблонов URL-адресов из blog.urls. URL-адрес post_detail определен в файле urls.py приложения blog.

Результирующий строковый литерал, blog:post_detail, можно использовать глобально в проекте, чтобы ссылаться на URL-адрес детальной информации о посте. Этот URL-адрес имеет обязательный параметр – id извлекаемого поста блога.

Идентификатор id объекта Post был включен в качестве позиционного аргумента, используя параметр args=[self.id].

УСОВЕРШЕНСТВОВАНИЕ БЛОГА ЗА СЧЕТ ПРОДВИНУТЫХ ФУНКЦИОНАЛЬНОСТЕЙ

Давайте заменим URL-адреса детальной информации о посте в шаблонах новым методом get_absolute_url(). Отредактируйте файл blog/post/list.html, заменив строку

```
<a href="{% url 'blog:post_detail' post.id %}">
```

```
<a href="{{ post.get_absolute_url }}">
```

УСОВЕРШЕНСТВОВАНИЕ БЛОГА ЗА СЧЕТ ПРОДВИНУТЫХ ФУНКЦИОНАЛЬНОСТЕЙ

Теперь файл blog/post/list.html должен выглядеть следующим образом:

```
{% extends "blog/base.html" %}

{% block title %}My Blog{% endblock %}

{% block content %}
  <h1>My Blog</h1>
  {% for post in posts %}
    <h2>
      <a href="{{ post.get_absolute_url }}">
        {{ post.title }}
      </a>
    </h2>
    <p class="date">
      Published {{ post.publish }} by {{ post.author }}
    </p>
    {{ post.body|truncatewords:30|linebreaks }}
  {% endfor %}
{% endblock %}
```

УСОВЕРШЕНСТВОВАНИЕ БЛОГА ЗА СЧЕТ ПРОДВИНУТЫХ ФУНКЦИОНАЛЬНОСТЕЙ

Откройте оболочку и исполните следующую ниже команду, чтобы запустить сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу http://127.0.0.1:8000/blog/ в своем браузере. Ссылки на одиночные посты блога по-прежнему должны работать. Теперь Django формирует их, используя метод get_absolute_url() модели Post.

СОЗДАНИЕ ДРУЖЕСТВЕННЫХ ДЛЯ ПОИСКОВОЙ ОПТИМИЗАЦИИ URL-АДРЕСОВ ПОСТОВ

Канонический URL-адрес представления детальной информации о посте блога в настоящее время выглядит как **/blog/1/**. Мы изменим шаблон URL-адреса, чтобы формировать дружественные для поисковой оптимизации URL-адреса постов. В целях формирования URL-адресов одиночных постов мы будем использовать дату публикации **publish** и значения **slug**.

Присоединив даты, мы приведем URL-адрес детальной информации о посте к следующему виду: **/blog/2022/1/1/who-was-django-reinhardt/**. Мы предоставим поисковым механизмам дружественные для индексации URL-адреса, содержащие как заголовок, так и дату поста.

СОЗДАНИЕ ДРУЖЕСТВЕННЫХ ДЛЯ ПОИСКОВОЙ ОПТИМИЗАЦИИ URL-АДРЕСОВ ПОСТОВ

Для того чтобы получить одиночные посты с комбинацией даты публикации и слага, необходимо обеспечить, чтобы ни одну запись невозможно было сохранить в базе данных с тем же значением поля **slug** и поля **publish**, что и у существующего поста.

Мы предотвратим хранение в модели **Post** дублирующихся записей, определив, что слаги являются уникальными для даты публикации поста. Отредактируйте файл models.py, добавив следующий ниже параметр **unique_for_date** в поле **slug** модели **Post**:

```
class Post(models.Model):
    # ...
    slug = models.SlugField(max_length=250,
                            unique_for_date='publish')
    # ...
```

СОЗДАНИЕ ДРУЖЕСТВЕННЫХ ДЛЯ ПОИСКОВОЙ ОПТИМИЗАЦИИ URL-АДРЕСОВ ПОСТОВ

Теперь при использовании параметра **unique_for_date** поле **slug** должно быть уникальным для даты, сохраненной в поле **publish**. Обратите внимание, что поле **publish** является экземпляром класса **DateTimeField**, но проверка на уникальность значений будет выполняться только по дате (не по времени).

Django будет предотвращать сохранение нового поста с тем же именем, что и у существующего поста на заданную дату публикации. В результате мы обеспечили уникальность слагов для даты публикации, поэтому теперь можно извлекать одиночные посты по полям **publish** и **slug**. Мы изменили модели, поэтому давайте создадим миграции. Обратите внимание, что параметр **unique_for_date** не соблюдается на уровне базы данных, поэтому миграция базы данных не требуется. Между тем миграции в Django используются для отслеживания всех изменений модели. Мы создадим миграцию только для того, чтобы привести миграции в соответствие с текущим состоянием модели.

СОЗДАНИЕ ДРУЖЕСТВЕННЫХ ДЛЯ ПОИСКОВОЙ ОПТИМИЗАЦИИ URL-АДРЕСОВ ПОСТОВ

Выполните следующую ниже команду в командной оболочке:

```
python manage.py makemigrations blog
```

Вы должны получить следующий ниже результат:

```
Migrations for 'blog':
    blog/migrations/0002_alter_post_slug.py
    - Alter field slug on post
```

Django только что создал файл 0002_alter_post_slug.py внутри каталога migrations приложения blog.

СОЗДАНИЕ ДРУЖЕСТВЕННЫХ ДЛЯ ПОИСКОВОЙ ОПТИМИЗАЦИИ URL-АДРЕСОВ ПОСТОВ

Выполните следующую ниже команду в командной оболочке, чтобы применить существующие миграции:

```
python manage.py migrate
```

Вы получите результат, который заканчивается такой строкой:

```
Applying blog.0002_alter_post_slug... OK
```

Django будет считать, что все миграции были применены и модели синхронизированы. В базе данных не будет выполнено никаких действий, поскольку параметр **unique_for_date** не применяется на уровне базы данных.

ВИДОИЗМЕНЕНИЕ ШАБЛОНОВ URL-АДРЕСОВ

Давайте видоизменим шаблоны URL-адресов, чтобы использовать дату публикации и слаг для URL-адреса детальной информации о посте. Отредактируйте файл **urls.py** приложения **blog**, заменив строку

```
path('<int:id>/', views.post_detail, name='post_detail'),
```

строками

```
path('<int:year>/<int:month>/<int:day>/<slug:post>/',
         views.post_detail,
         name='post_detail'),
```

ВИДОИЗМЕНЕНИЕ ШАБЛОНОВ URL-АДРЕСОВ

Теперь файл **urls.py** должен выглядеть следующим образом:

```
from django.urls import path
from . import views

app_name = 'blog'

urlpatterns = [
    # представления поста
    path('', views.post_list, name='post_list'),
    path('<int:year>/<int:month>/<int:day>/<slug:post>/',
         views.post_detail,
         name='post_detail'),
]
```

ВИДОИЗМЕНЕНИЕ ШАБЛОНОВ URL-АДРЕСОВ

Шаблон URL-адреса представления **post_detail** принимает следующие ниже аргументы:

- **year**: требуется целое число;
- **month**: требуется целое число;
- **day**: требуется целое число;
- **post**: требуется слаг (строка, содержащая только буквы, цифры, знаки подчеркивания или дефисы).

Конвертор пути **int** используется для параметров **year**, **month** и **day**, тогда как конвертор пути **slug** применяется для параметра **post**. В предыдущей главе вы узнали о конверторах путей

ВИДОИЗМЕНЕНИЕ ПРЕДСТАВЛЕНИЙ

Теперь необходимо видоизменить параметры представления **post_detail**, чтобы они соответствовали новым параметрам URL-адреса, и использовать их для извлечения соответствующего объекта **Post**.
Откройте файл **views.py** и отредактируйте представление **post_detail**, как показано ниже:

```
def post_detail(request, year, month, day, post):
    post = get_object_or_404(Post,
                             status=Post.Status.PUBLISHED,
                             slug=post,
                             publish__year=year,
                             publish__month=month,
                             publish__day=day)
    return render(request,
                  'blog/post/detail.html',
                  {'post': post})
```

ВИДОИЗМЕНЕНИЕ ПРЕДСТАВЛЕНИЙ

Мы видоизменили представление **post_detail**, чтобы использовать аргументы **year**, **month**, **day** и **post** и извлекать опубликованный пост с заданным слагом и датой публикации. Ранее, добавив в поле **slug** значение параметра **unique_for_date='publish'** модели **Post**, мы обеспечили, чтобы был только один пост со слагом на заданную дату. Таким образом, используя дату и слаг, можно извлекать одиночные посты.

```
def post_detail(request, year, month, day, post):
    post = get_object_or_404(Post,
                             status=Post.Status.PUBLISHED,
                             slug=post,
                             publish__year=year,
                             publish__month=month,
                             publish__day=day)
    return render(request,
                  'blog/post/detail.html',
                  {'post': post})
```

ВИДОИЗМЕНЕНИЕ КАНОНИЧЕСКОГО URL-АДРЕСА ПОСТОВ

Также необходимо видоизменить параметры канонического URL-адреса для постов блога, чтобы они сочетались с новыми параметрами URL-адреса. Откройте файл **models.py** приложения **blog** и отредактируйте метод **get_absolute_url()**, как показано ниже:

```
class Post(models.Model):
    # ...
    def get_absolute_url(self):
        return reverse('blog:post_detail',
                       args=[self.publish.year,
                             self.publish.month,
                             self.publish.day,
                             self.slug])
```

Запустите сервер разработки, набрав следующую ниже команду в командной оболочке:

```
python manage.py runserver
```

ВИДОИЗМЕНЕНИЕ КАНОНИЧЕСКОГО URL-АДРЕСА ПОСТОВ

Далее можно вернуться в свой браузер и кликнуть по одному из заголовков постов, чтобы посмотреть детальную информацию о посте. Вы должны увидеть что-то вроде этого:

Страница представления детальной информации о посте

Взгляните на URL-адрес – он должен выглядеть как **/blog/2022/1/1/who-wasdjango-reinhardt/**. Вы разработали дружественные для поисковой оптимизации URL-адреса постов блога.

ДОБАВЛЕНИЕ ПОСТРАНИЧНОЙ РАЗБИВКИ

Когда вы начнете добавлять контент в свой блог, вы сможете легко хранить десятки и даже сотни постов в своей базе данных. Возможно, вы захотите разделить список постов на несколько страниц, не отображая все записи на одной странице, и вставить навигационные ссылки на разные страницы. Эта функциональность называется постраничной разбивкой, и ее можно найти почти в каждом веб-приложении, которое показывает длинные списки элементов. Например, Google использует постраничную разбивку с целью распределения результатов поиска по нескольким страницам. На рис. показаны постранично разбитые ссылки Google на страницы результатов поиска:

Постранично разбитые ссылки Google на страницы результатов поиска

В Django есть встроенный класс постраничной разбивки, который позволяет легко управлять постранично разбитыми данными, при этом имеется возможность определять число объектов, которое необходимое возвращать в расчете на страницу, и извлекать записи, соответствующие запрошенной пользователем странице.

ДОБАВЛЕНИЕ ПОСТРАНИЧНОЙ РАЗБИВКИ В ПРЕДСТАВЛЕНИЕ СПИСКА ПОСТОВ

Отредактируйте файл **views.py** приложения **blog**, импортировав класс **Django Paginator** и видоизменив представление **post_list**, как показано ниже:

```
from django.shortcuts import render, get_object_or_404
from .models import Post
from django.core.paginator import Paginator

def post_list(request):
    post_list = Post.published.all()
    # Постраничная разбивка с 3 постами на страницу
    paginator = Paginator(post_list, 3)
    page_number = request.GET.get('page', 1)
    posts = paginator.page(page_number)
```

```
    return render(request,
                  'blog/post/list.html',
                  {'posts': posts})
```

ДОБАВЛЕНИЕ ПОСТРАНИЧНОЙ РАЗБИВКИ В ПРЕДСТАВЛЕНИЕ СПИСКА ПОСТОВ

Давайте рассмотрим новый исходный код, который был добавлен в представление.

1. Мы создаем экземпляр класса **Paginator** с числом объектов, возвращаемых в расчете на страницу. Мы будем отображать по три поста на страницу.
2. Мы извлекаем **HTTP GET**-параметр **page** и сохраняем его в переменной **page_number**. Этот параметр содержит запрошенный номер страницы. Если параметра **page** нет в **GET**-параметрах запроса, то мы используем стандартное значение 1, чтобы загрузить первую страницу результатов.
3. Мы получаем объекты для желаемой страницы, вызывая метод **page()** класса **Paginator**. Этот метод возвращает объект **Page**, который хранится в переменной **posts**.
4. Мы передаем номер страницы и объект **posts** в шаблон.

СОЗДАНИЕ ШАБЛОНА ПОСТРАНИЧНОЙ РАЗБИВКИ

Далее необходимо создать навигацию по страницам, чтобы пользователи имели возможность просматривать разные страницы. Мы создадим шаблон отображения постранично разбитых ссылок и сделаем его типовым, чтобы иметь возможность реиспользовать шаблон для постраничной разбивки любого объекта на веб-сайте. Внутри каталога **templates/** создайте новый файл и назовите его **pagination. html**. Добавьте в файл следующий ниже исходный код **HTML**:

```
<div class="pagination">
    <span class="step-links">
      {% if page.has_previous %}
        <a href="?page={{ page.previous_page_number }}">Previous</a>
      {% endif %}
      <span class="current">
        Page {{ page.number }} of {{ page.paginator.num_pages }}.
      </span>
      {% if page.has_next %}
        <a href="?page={{ page.next_page_number }}">Next</a>
      {% endif %}
    </span>
  </div>
```

СОЗДАНИЕ ШАБЛОНА ПОСТРАНИЧНОЙ РАЗБИВКИ

Это типовой шаблон постраничной разбивки. Предполагается, что данный шаблон будет иметь в контексте объект **Page**, чтобы прорисовывать предыдущую и следующую ссылки, а также отображать текущую страницу и общее число страниц результатов. Давайте вернемся к шаблону **blog/post/list.html** и разместим шаблон pagination.html в нижней части блока **{% content %}**, как показано ниже:

```
{% extends "blog/base.html" %}

{% block title %}My Blog{% endblock %}

{% block content %}
  <h1>My Blog</h1>
  {% for post in posts %}
    <h2>
      <a href="{{ post.get_absolute_url }}">
        {{ post.title }}
      </a>
    </h2>
    <p class="date">
      Published {{ post.publish }} by {{ post.author }}
    </p>
    {{ post.body|truncatewords:30|linebreaks }}
  {% endfor %}
  {% include "pagination.html" with page=posts %}
{% endblock %}
```

СОЗДАНИЕ ШАБЛОНА ПОСТРАНИЧНОЙ РАЗБИВКИ

Шаблонный тег **{% include %}** загружает данный шаблон и прорисовывает его с использованием текущего контекста шаблона. Ключевое слово **with** используется для того, чтобы передавать дополнительные контекстные переменные в шаблон.

Для прорисовки в шаблоне постраничной разбивки используется переменная **page**, при этом объект **Page**, который мы передаем из представления в шаблон, называется **posts**. Мы используем выражение **with page=posts**, чтобы передавать переменную, ожидаемую шаблоном постраничной разбивки.

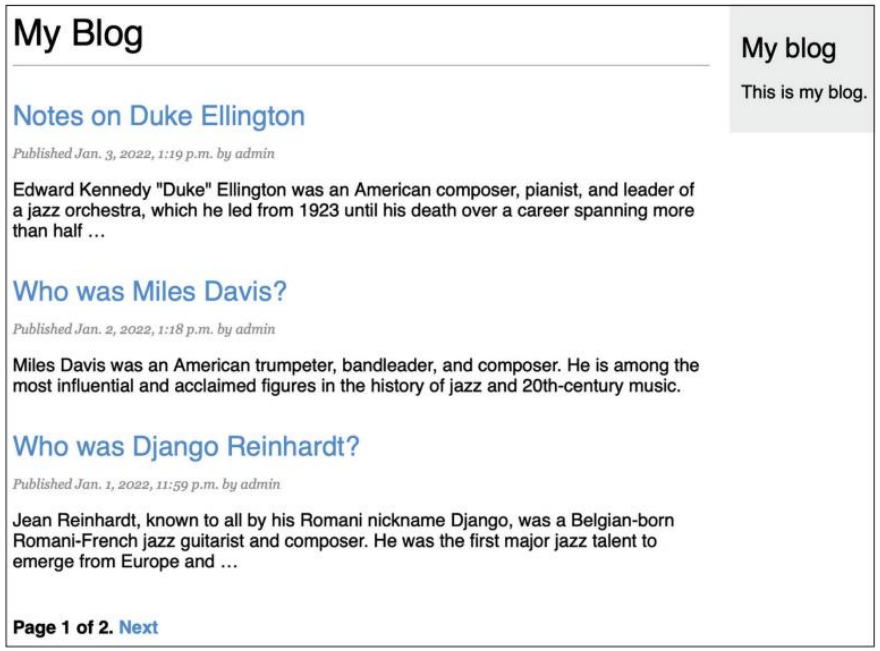
Описанному методу можно следовать для применения шаблона постраничной разбивки для любого типа объекта. Запустите сервер разработки, набрав следующую ниже команду в командной оболочке:

```
python manage.py runserver
```

СОЗДАНИЕ ШАБЛОНА ПОСТРАНИЧНОЙ РАЗБИВКИ

Пройдите по URL-адресу **http://127.0.0.1:8000/admin/blog/post/** в своем браузере и используйте сайт администрирования, чтобы создать в общей сложности четыре разных поста. Проверьте, чтобы у всех этих постов был установлен статус **Published**.

Теперь пройдите по URL-адресу **http://127.0.0.1:8000/blog/** в своем браузере. Вы должны увидеть первые три поста в обратном хронологическом порядке, а затем навигационные ссылки в нижней части списка постов, как показано ниже:



Страница списка постов с постраничной разбивкой ссылок внизу

СОЗДАНИЕ ШАБЛОНА ПОСТРАНИЧНОЙ РАЗБИВКИ

Если кликнуть по **Next (Далее)**, то можно увидеть последний пост. URLадрес второй страницы содержит **GET-параметр ?page=2**. Указанный параметр используется представлением для загрузки запрошенной страницы результатов с использованием постраничного разбивщика.

Вторая страница результатов